SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. **10**, 225–247 (1998)

Research

Incremental Delivery Reduces Maintenance Cost: A COCOMO-based Study

PEI HSIA*, CHIH-TUNG HSU, DAVID CHENHO KUNG and ERIC J. BYRNE

Software Engineering Center for Telecommunication, Department of Computer Science and Engineering, The University of Texas at Arlington, Box 19015, Arlington TX 76019–0015, U.S.A.

SUMMARY

Incremental delivery (ID) is a software development paradigm which advocates that systems be delivered to end users in usable, useful and semi-independent increments. ID differs from the traditional development paradigm, which we call monolithic delivery (MD), in which a software system is considered as a monolithic, inseparable whole delivered as one unit. The purpose of this study is to compare the ID and MD approaches in terms of their maintenance efforts through an analytical parametric study. The results of the study provide insight into how incremental delivery can be employed to reduce software maintenance effort. © 1998 John Wiley & Sons, Ltd.

KEY WORDS: incremental development; incremental delivery; software maintenance; COCOMO model; cost estimation

1. INTRODUCTION

Traditionally, software systems have been considered monolithically: no part is separable from the rest, and all software components must be present to achieve an operational system. Recently, researchers and practitioners have been advocating incremental development, and even incremental delivery, to construct and deliver software systems (Abbott, 1997 p.88–94; Gilb, 1988, p.84; Hough, 1993; Hsia, Yaung and Jiam, 1986; Mills, Dyer and Linger, 1987; Linger, 1994; Pittman, 1993; Tran and Galka, 1991). Incremental delivery takes incremental development a step further by delivering software systems to end users in increments. Each delivered increment supports a partial set of requirements, and its functionality is visible to the end user. These approaches are conceptually appealing because they provide both customers and management with an essential ingredient that is conspicuously absent in the MD approach: progress visibility. The delivered increments can be examined by the end user for their functionality, and they provide a foretaste of

* Correspondence to: Pei Hsia, Software Engineering Center for Telecommunication, Department of Computer Science and Engineering, The University of Texas at Arlington, Box 19015, Arlington TX 76019–0015, U.S.A. Email: hsia@cse.uta.edu

CCC 1040-550X/98/040225–23$17.50

Received 2 November 1996
Revised 16 March 1998

the things to come. At the same time, the delivered increments serve as concrete progress achievements to customers, management and system developers themselves.

Incremental delivery transforms the major steps in software development, such as requirements analysis, system partitioning and system integration, into a user-centred perspective. ID partitions a whole system from its utility point of view (i.e., the external view). The external view allows a system to be divided into usable increments. Each increment is semi-independent from the rest of the system and can be developed and delivered to the user in a well-defined sequence according to priority, precedence relation and other criteria. The delivered increments are immediately usable to the customer to fulfil part of his/her mission.

A key issue to the success of ID is to cluster increments in such a way that each increment can operate without much functional help from other increments. However, only under rare situations can a system be partitioned into mutually independent increments. Reasons for this include: (1) functional interdependence between increments, (2) interconnectedness by shared data structures or environmental entities, (3) arrangement of the delivery of increments in proper order, and (4) the nature of the problem itself. Thus, in ID development, more or less *extra code* is needed to achieve the semi-independence of an increment and to make each increment self-contained.

The introduction of the extra code increases the size of the system, which results in an increase in maintenance effort and cost. However, the disadvantages of increasing the size of a software system are neutralized—in fact, outweighed—by the many benefits of ID. According to a standard procedure proposed by Schneidewind (1989), ID is effective from the perspective of software maintenance. Regardless of the methodologies that are employed to implement the increments, the 'side-effects' of maintenance are always limited only to the increments being maintained. ID facilitates 'selective change', since any change to an increment does not affect the normal functionality of other increments (i.e., the rest of the system). Any change in an ID system is localized to 'small' increments. Indeed, each delivered increment has a built-in 'firewall' around itself (Leung and White, 1990).

The objectives of this study are to investigate three key questions:

(1) How much better is ID over MD from the perspective of maintenance effort?
(2) What is the ideal range for the number of increments of an ID system?
(3) How much extra code is allowed if ID is to be beneficial compared with MD in the context of software maintenance?

The motivations behind this study are: (1) to emphasize the little-recognized benefits of ID on maintenance effort; (2) to pave the way for future comparative studies between ID and MD system maintenance experimentally; and (3) to extract useful information to help guide future ID research.

The next section reviews incremental delivery and briefly discusses COCOMO and its newest version, COCOMO 2.0. Section 3 describes the characteristics of maintaining an incrementally delivered system through a simple hypothetical example. Section 4 presents a mathematical equation and derives ID maintenance effort estimation based on the equation. The results of the study are presented in Section 5, in which the effects of the

parameters on maintenance effort are examined, and the results are compared with those of monolithic development. Section 6 summarizes our findings and outlines the direction of future research.

2. BACKGROUND

2.1. Incremental delivery

Incremental delivery can be broadly classified into two categories: expansive incremental delivery and partitioned incremental delivery. The major difference between the two types is in the relationships among their delivered increments. The former continues to incorporate new capabilities into an already delivered increment and deliver it as a new delivery, while the latter provides additional functionality and is delivered as a separate increment.

If we use the highway analogy, the partitioned incremental delivery is like building one highway section at a time and putting it in use, while the expansive incremental delivery is like merging the new section into the previous chunk to build a longer chunk and replacing the already delivered chunk with the new and longer chunk of highway when it is done. In one sense, the expansive type develops a more integrated system, since all the functions are fused together into one system, and the partitioned type produces the system in semi-related chunks. However, their benefits cannot be observed simply this way.

The former type considers the incremental delivery paradigm only as a mechanism to transform an invisible process into a stepwise visible process. The latter considers incremental delivery, in addition to the transformation process, as a system-partitioning tool. That tool implants in the software product extra desirable characteristics for future life cycle activities, such as modification, extension, reduction and all related testing. If the system is designed with a *firewall* boundary between any two increments, then it will provide tremendous savings to all related software maintenance activities.

Obviously, the partitioned incremental delivery is more advantageous than the expansive type. How does one decide to go with which types of incremental delivery? This decision must be made in the very beginning of a project. Otherwise, there will be no choice left but to do the expansive type. There are several reasons that the partitioned type can be selected: (1) investigate in the early stage of a project and decide that it is feasible to adopt the partitioned incremental model; (2) obtain strong support from management in both the user's and developer's organizations; (3) obtain commitment to use this model from all the developers; (4) start with a workshop to describe the process of the partitioned incremental delivery model to enable all of the professionals in the project to develop a similar understanding and regularly maintain the workshop and use it to make known the status of the project. If these steps were not considered at the outset, it would be very difficult to carry out the partitioned incremental delivery in the project.

Currently, most of the incremental approaches that are practised in industry fall into the first category. Examples include Gilb's *Evolutionary Delivery* (Gilb, 1988, p.84), IBM's *Cleanroom Process* (Mills, Dyer and Linger, 1987; Linger, 1994), Hough's *Rapid Delivery* (Hough, 1993), and Xerox's *Chunking Method* (Abbott, 1997, p.88–94).

Evolutionary delivery

Evolutionary delivery starts with identifying a set of high-level evolutionary steps based on a list of functional requirements and a set of alternative-solution specifications. The initial steps are sorted into a sequence of development activities based on the user value to development cost (or risk) ratio. Each step is iteratively broken down into smaller delivery steps, usually one to five percent of the total project effort. Steps with the highest ratio are delivered first.

Cleanroom process

The cleanroom process is based on incrementally developing and certifying a pipeline of software increments. The functional content and the development sequence of each increment is determined based on the functional specification. In parallel, a system usage scenario that serves as a basis for generating test cases is also prepared.

The development of each increment typically goes through a sequence of design and verification cycles. Each newly completed increment is added to the system by assimilating it into existing increments. The system as a whole is then delivered to the certification team for statistical usage testing and quality certification using the test cases generated from the usage scenario specification.

The cleanroom process relies on stable system requirements to set up the incremental development plan. New requirements arising from user experience are incorporated in the next version. The incremental development plan is re-scheduled by incorporating new requirements. In fact, system functionality grows incrementally to match the user's expectation.

System integration follows the strategy similar to that of top–down integration testing. Early increments implement system architecture and defer local implementation with stubs. As each subsequent increment is completed, stubs in previous increments are replaced by real operational code. The cleanroom process is incremental in that system development is achieved in increments. However, the delivery of the system is still monolithic.

Rapid delivery

Rapid delivery begins with collecting sufficient high-level requirements to set up the development plan. A suitable application segmentation strategy is developed by analysing the high-level requirements. The development of each application increment then goes through an iterative process of additional requirements gathering, design, coding and testing.

Like the cleanroom process, rapid delivery follows a top–down approach to application development. Early increments define the overall application architecture and partially implement the portion which is needed to support the development of subsequent increments. The remaining parts of the application that will be developed later are filled by stub functions or modules. As each new increment is developed, the associated stubs are removed from the previous increments, and the new increment itself is integrated as a part of the total application.

Chunking method

Chunking decomposes software into sets of close-related, customer-visible features, and then develops these 'feature-sets' in six-to-eight week 'chunks' of work (Abbott, 1994). Within a chunk, a team of four to six developers proposes a concept for the work to be performed in the chunk. Upon approval of the concept, the team develops a specification, and then designs, implements, unit tests and delivers the software to QA for integration and testing. Every chunk delivers a self-consistent feature set to QA; therefore, it is possible to release partial interim results to the user on demand. However, the delivery is monolithic because chunks developed by different teams must be integrated at the QA department. Chunking is a resource-driven approach in that it adjusts functionality to meet available schedules and resources.

2.2. The COCOMO models

The 1981 COCOMO

The COCOMO model is probably the most comprehensive and well-documented of all software cost estimation models. It was derived from a set of 63 software projects at TRW (Boehm, 1981a, p.75). COCOMO is designed to predict the required development and maintenance effort based on estimated software size. It includes a hierarchy of three levels of progressively complete models: basic, intermediate and detailed. The basic model computes development effort and cost based on the estimated thousands of lines of delivered source instructions (KDSI). The intermediate model considers a set of 15 cost drivers besides the estimated software size. The detailed model extends the intermediate model with an assessment of the impact of the cost drivers on each phase of the software life cycle.

COCOMO categorizes software projects into three modes: (1) organic mode—a small team of application-experienced people working on a familiar project; (2) semi-detached mode—an intermediate project in which a mix of experienced and less-experienced people work on projects with less-than-rigid requirements; and (3) embedded mode—a project with rigid interface and operational constraints. Each mode of the project is associated with a formula for the estimation of development and maintenance effort. The formulas of the basic COCOMO model are shown in Table 1. Our study is based on the basic COCOMO annual software maintenance cost model:

$$(MM)_{\mathrm{AM}} = 1{\cdot}0\ (ACT)\ (MM)_{\mathrm{D}}$$

Table 1. The basic COCOMO estimates of software development effort

Mode	Equation
Organic	$(MM)_{\mathrm{D}} = 2{\cdot}4(\mathrm{KDSI})^{1{\cdot}05}$
Semi-detached	$(MM)_{\mathrm{d}} = 3{\cdot}0(\mathrm{KDSI})^{1{\cdot}/12}$
Embedded	$(MM)_{\mathrm{D}} = 3{\cdot}6(\mathrm{KDSI})^{1{\cdot}20}$

where $(MM)_{AM}$ is the annual maintenance effort in man-months; and *ACT* is annual change traffic, the fraction of the software product's source instructions which undergo change during a year, either through addition or modification (Boehm, 1981a, p.71).

COCOMO 2.0

The increasing diversity of new software development processes and capabilities has created new challenges to software cost estimation. These include non-sequential and rapid-development process models; reuse-driven approaches involving commercial off-the-shelf packages, re-engineering and applications generation capabilities; object-orientated approaches supported by middleware; and software process maturity initiatives (Boehm *et al*., 1995). COCOMO 2.0 is currently being developed and will address the various types of new development approaches mentioned above.

Major features of COCOMO 2.0 include a tailorable mix of variable-granularity sizing models, involving object points, function points and source lines of code; a non-linear estimation model for software reuse and re-engineering; a hyper-linear scaling model for modelling the relative software diseconomies of scale (Conte, Dunsmore and Shen, 1986, p.283; Banker and Kemerer, 1989); and revisions to the original COCOMO effort-multiplier cost drivers.

The maintenance effort estimation is handled by the *Reuse Model*, in which maintenance effort is estimated as:

$$PM = A \times (ESLOC)^B$$

where A is a constant; B is the scaling factor; and *ESLOC*, as determined by the following equation, is the equivalent source lines of code to be used as the size measure for maintenance effort estimation.

$$ESLOC = ASLOC \times \frac{AA + SU + 0{\cdot}4 \times DM + 0{\cdot}3 \times CM + 0{\cdot}3 \times IM}{100}$$

Note that the derivation of *ESLOC* involves the estimation of the amount of software to be adapted (*ASLOC*) and five other parameters: the percentage of design modification (*DM*), the percentage of code modification (*CM*), and the percentage of the original integration effort required for integrating the reused software (Boehm *et al*., 1995), the degree of assessment and assimilation needed to determine whether it is appropriate to reuse a software module and the ease of integrating its description into the overall product description (*AA*), and the ease with which the software can be understood (*SU*). The effects of these parameters in sizing the maintenance effort will not be further studied. Instead, we will use a single parameter, size adjustment parameter or *SAP*, to represent them.

The value of the scaling factor B is derived by rating (on a 0-to-5 scale) and summing across five factors (W_i)—precedentedness, development flexibility, architecture/risk resolution, team cohesion and process maturity. That is

$$B = 1{\cdot}01 + 0{\cdot}01 \sum W_i$$

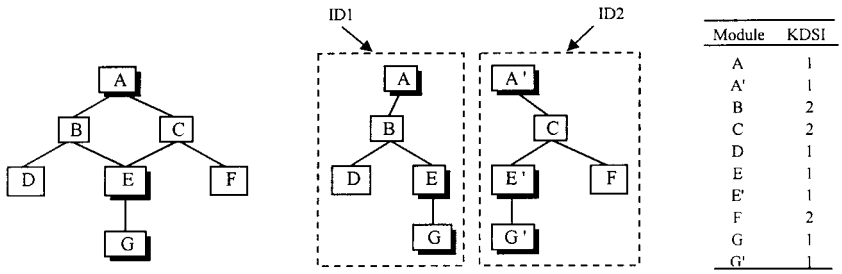
Note that the minimum (best case) and maximum (worst case) values of B are 1·01 and 1·26, respectively.

3. MAINTENANCE OF AN INCREMENTALLY DELIVERED SYSTEM

A simple example will be presented to highlight the process of maintaining an ID system. Suppose that a software structure chart of a simple hypothetical system is given, as shown in Figure 1(a). One of the many possible ID implementations is to partition the system from its utility view and build it in two increments, ID1 and ID2 (Figure 1(b)). To partition an intra-related system into two semi-independent increments, a duplication of some of the functionalities may be needed. Assume that ID1 is delivered to the customer before ID2. Modules A, B, D, E and G were developed and integrated in the delivery of ID1. ID2 needs the functionalities provided by Modules A, E and G to be operational and self-contained. ID2 may reuse Modules A, E and G or implement the functionalities anew; let them be A′, E′ and G′. These three modules are extra, because they are not needed if the system is delivered in the conventional manner.

Assume that the size of each module in KDSI is defined as in Figure 1(c). Thus, ID1 (A + B + D + E + G) has six KDSI, and ID2 (A′ + C + E′ + F + G′) has seven KDSI. Suppose a repair of Module E (or E′) is requested. Assume that 20% of the lines of code in Module E (or E′) need to be changed. Two possible situations may occur: (I) both ID1 and ID2 need the change (this may occur when the requirements are changed), or (II) only one increment, say ID1, needs the change. In Case I, two possible situations can occur: (1) Module E is reused in ID2 (i.e., E′ = E), or (2) Module E′ is a different implementation from Module E.

In Case I.1, although Module E is modified in both increments, the effects of the modification in ID1 will not ripple across its boundary to affect the normal functionality of ID2 and vice versa. Indeed, each delivered increment has a built-in 'firewall' around itself. In Case I.2, the modification is totally independent. The annual maintenance effort of ID1 is $(0{\cdot}2/6)(a \times 6^b)$ and that of ID2 is $(0{\cdot}2/7)(a \times 7^b)$, where a and b are constants of the basic COCOMO equations. In the MD system, the maintenance effort is



Module	KDSI
A	1
A'	1
B	2
C	2
D	1
E	1
E'	1
F	2
G	1
G'	1

Figure 1. (a) Software structure chart of a simple hypothetical system; (b) one possible ID implementation; (c) assumed size of each module in the hypothetical system

$(0{\cdot}2/10)(a \times 10^b)$. As one can see from Table 2, the total maintenance effort of the ID system is almost twice that of the MD system. This indicates that the maintenance effort of an ID system is proportional to the number of increments. In reality, this argument is erroneous for at least two reasons. First, the difficult part of software maintenance is not this semi-mechanical work of modification but 'locating the defects', 'deciding a new design', and/or identifying the parts that need to be changed when ported to a new environment. Second, as will be shown next, the amount of code similarity, in the case of code reuse, will decrease as the maintenance effort becomes more localized.

In Case II, the change is required only in ID1, and the maintenance effort of the entire ID system is $(0{\cdot}2/6)(a \times 6^b) + 0$ (no change to ID2), a little less than that of the MD system. After the change, Module E is no longer considered as identical to Module E′, unless the same change is judged essential to ID2 in the future. For future maintenance, the work becomes more and more localized. As the process continues, Case II becomes dominant.

In Case I, as already noted, the change to ID1 and ID2 is really the same. Hence, the annual maintenance effort of the whole ID system is equal to either ID1 or ID2, not their sum. The real effort is determined by the order of maintenance. For example, if ID1 is repaired before ID2, the maintenance effort will be 0·52 man-months for an organic mode system (see Table 2).

4. PARAMETRIC STUDY

In the fields of civil and mechanical engineering, parametric studies—analytical or experimental—are often conducted to investigate the physical properties of an artefact. They help civil engineers understand the impact of the lateral-loading position of vehicles on multi-girder bridges and help mechanical engineers reveal the influence of elevated temperature on the compressive strength of composite materials, and so on (Mabson *et al*., 1984; Miyano *et al*., 1986). In analytical studies, the impact factors are identified and modelled. The effects of the impact factors are then examined and interpreted based on the model. Comparable experimental studies may also be conducted to validate the analytical results.

In the software community, experimental studies are used either to compare two different development approaches (Alavi, 1984; Basili and Reiter, 1981; Boehm, Gray and Seewaldt, 1984) or to study the effects of a particular development methodology on software quality

Table 2. A comparison of software maintenance effort

	ID (13 DKSI)				
Mode	ID1 (6 DKSI)	ID2 (7 KDSI)	Total effort (summed)	Real effort	MD (10 DKSI)
Organic	0·52	0·53	1·05	0·52 or 0·53	0·54
Semi-detached	0·74	0·76	1·50	0·74 or 0·76	0·79
Embedded	1·03	1·06	2·09	1·03 or 1·06	1·14

and/or productivity (Boehm, 1981b; Buck and Dobbins, 1984; Eckhard *et al.*, 1991). The results generated from a well-designed and controlled experiment are then interpreted, and facts and/or hypotheses are confirmed.

Conducting non-trivial experiments is usually costly and impractical because of the characteristics of the problem itself. It is infeasible for a new development paradigm, like ID, to have a set of empirical data from maintenance. However, data collection will be an important activity for justifying the effectiveness of a particular approach. The lack of real-world data motivated our attempt to analytically study the effectiveness of ID from the perspective of software maintenance.

To provide a more concrete numerical evidence showing that ID is beneficial, the 1981 COCOMO model and the newest version of COCOMO—COCOMO 2.0—are used to derive sample data for comparison.

4.1. A distribution equation

The distributions of the parameters among the increments (e.g., increment size and the size of the extra code) are not available and are expected to have a highly irregular pattern. The situation is worse when we consider two or three parameters together. The *order*† of the increments and their associated parameters needs to be assumed before we can model them.

However, if we preserve their orders, one single distribution equation might not be enough to model closely the behaviour of their parameters. One simple way is to pre-order the increments according to their relative value of one single parameter (called the major parameter) in which we are interested (for example, the size of the extra code). The increments are ordered into a *bidirectionally increasing* sequence according to the relative value of the major parameter. For example, we may order the set {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} into a sequence ⟨1, 3, 5, 7, 9, 10, 8, 6, 4, 2⟩. Notice that the sequence is bidirectionally increasing from both ends of the sequence. As we will see, one interesting characteristic of this ordering scheme is that we can always produce a bell-shaped curve—a highly regular pattern of distribution. This allows us to model numerous types of irregular distributions with one single mathematical equation. The distribution equation is defined as:

$$y(i) = \begin{cases} \dfrac{1}{\sqrt{(1-x^2)^2 + L^2x^2}} & \text{if } 1 \leqslant i < p \\ \dfrac{1}{\sqrt{(1-x^2)^2 + R^2x^2}} & \text{if } p \leqslant i \leqslant n \end{cases} \tag{1}$$

† Note that here we are not considering the order of incremental delivery. The reason for ordering the increments is for the ease of employing the proposed distribution equation.

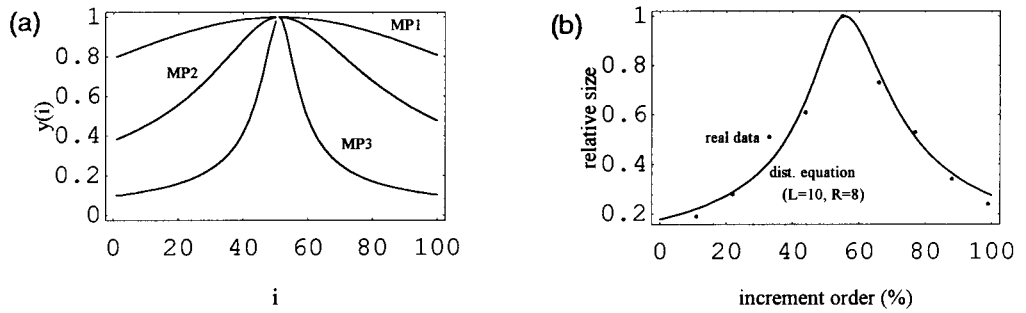


Figure 2. (a) Typical curves of the distribution equation; (b) the distribution of size can be closely modelled as the distribution equation

where $p = \lceil (n+1)/2 \rceil$ is the corresponding x-position of the peak (Figure 2(a)); $x = \dfrac{i-p}{n}$; and $L \geqslant 1{\cdot}41$ and $R \geqslant 1{\cdot}41$ are the constants which affect the shape of the curves on both sides of the peak, respectively. If $L = R$, then the curve will be symmetrical about the peak. By adjusting the values of L and R, we can closely model numerous kinds of projects. Some typical curves are shown in Figure 2(a). The meanings of the three curves—MP1, MP2 and MP3—will be explained later.

As indicated in Figure 2(a), the y-value of the *peak is always one.* This is the largest value of the objects we model. The values of the other objects are relative to the largest value, which is always less than or equal to one. Table 3 shows the data derived from an ID system (Jiam, 1985). As shown in Figure 2(b), the distribution equation demonstrates a good fit to the real data.

4.2. Parameter modelling

The equations we use to generate data for comparison are derived and described in Appendix B in which all the parameters are modelled in terms of the distribution equation.

Table 3. An ID system

Increment	Total size (LOC)	Total size (%)	Ordered sequence (i)	$y(i)$
1	907	22·6	5	1
2	167	4·3	1	0·19
3	687	16·4	6	0·73
4	480	12·0	7	0·53
5	301	7·6	8	0·34
6	218	5·5	9	0·24
7	250	6·3	2	0·28
8	462	11·5	3	0·51
9	552	13·8	4	0·61

To provide more complete and general results, we will consider three representative distributions to model the major parameters. They are:

(1) MP1 distribution: $y(i)$ with $L = 2$ and $R = 2$;
(2) MP2 distribution: $y(i)$ with $L = 5$ and $R = 4$;
(3) MP3 distribution: $y(i)$ with $L = 20$ and $R = 20$.

MP1 represents a uniform-like distribution, MP2 represents an average distribution, while MP3 intends to model a highly concentrated distribution. These three cases are illustrated in Figure 2(a).

To determine the effects of the major parameter, we need to control the behaviours of the other parameters (called supporting parameters). Since we have ordered the increments according to the relative value of the major parameter, the order of the supporting parameters must be adjusted accordingly. This ordering results in an unknown and highly irregular distribution pattern of the supporting parameters, which causes the failure of the distribution equation to model them. For example, when the effects of the distribution of the size of the extra code are desired, e_i will be our major parameter. The other parameter (i.e., k_i) will be the supporting parameter. Although e_i can be modelled well by the distribution equation, it fails to model k_i. The role of a supporting parameter, as its name suggests, is to integrate with the major parameter in a way that the effects of the major parameter on maintenance effort are well marked. The distribution type of the supporting parameter is not the major concern. For convenience, we will simply assume the supporting parameters are uniformly distributed. Table 4 lists the combinations of the major and supporting parameters we will consider in this study.

5. RESULTS

There are two reasonable yet conflicting predictions: (1) ID will reduce maintenance cost because smaller and separate increments are maintained, and (2) ID will increase maintenance cost because extra code is needed to achieve the semi-independence of each increment. This study will determine the trade-off between the two predictions. To develop an ID system that is more cost-effective than the comparable MD system, the largest allowed amount of extra code needs to be determined. We define cost-effective extra code ratio (CER) as a ratio of the total size of the extra code (E) relative to the total size of the non-extra code (K) in an ID system below which the ID system is more cost-effective

Table 4. Combinations of the major and supporting parameters

1981 COCOMO		COCOMO 2.0	
Major parameter	Supporting parameter	Major parameter	Supporting parameter
s_i	ACT_i	$ASLOC_i$	SAP_i
e_i	k_i	$\hat{e}_i$	SAP_i and $\hat{k}_i$

than the comparable MD system. A CER serves as a cordon at which ID can be beneficial if we can keep the value of the E/K ratio below this value. A CER of 0.5 indicates that the total size of the extra code is half of the total size of the non-extra code in an ID system.

We first assess research question 1: how much beter is ID over MD from the perspective of maintenance effort? As indicated in Figure 3(a)–(c), the results are that maintenance effort increases non-linearly with the size of the system; however, the increase of maintenance effort is less in ID systems than in MD systems if the value of E/K—the ratio of the total size of the extra code relative to the total size of the non-extra code—is below the CER. Here, the E/K ratio (0.2) is less than the CERs of semi-detached mode and embedded mode systems. In reality, as explained in Section 3, the maintenance of similar modules within different increments forces the increase of maintenance effort in ID systems even less than that shown in the figure.

Figures 4(b) and 4(c) show the substantial savings of maintenance effort in ID systems compared with MD systems. As indicated, the amount of effort saved increases non-linearly not only with the size but also with the mode of the system. For example, a 10-increment system 100 KDSI large, with 20% of the total extra code (i.e., $E/K = 0{\cdot}2$), will result in an annual saving of approximately four man-months of maintenance effort in

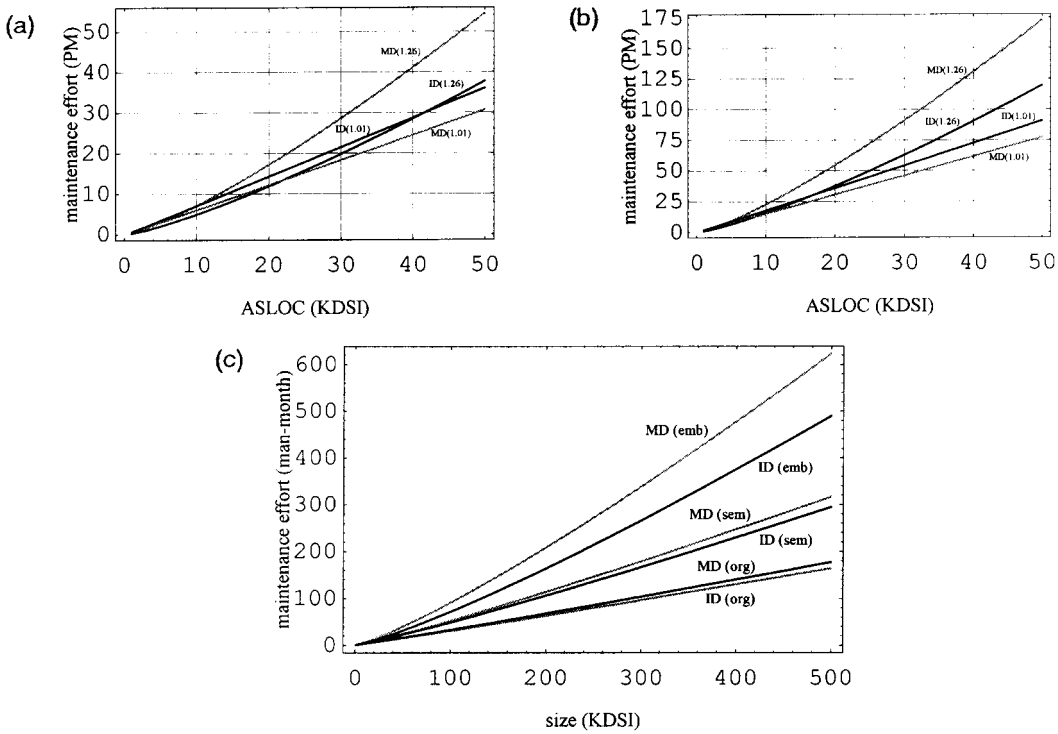


Figure 3. The relationship between maintenance effort and size: (a) COCOMO 2.0 with $n = 10$, $\hat{E}/\hat{K} = 0{\cdot}2$, $SAP_i = 0{\cdot}2$; *(b) COCOMO 2.0 with* $n = 10$, $\hat{E}/\hat{K} = 0{\cdot}2$, $SAP_i = 0{\cdot}5$; *(c) 1981 COCOMO with* $n = 10$, $E/K = 0{\cdot}2$

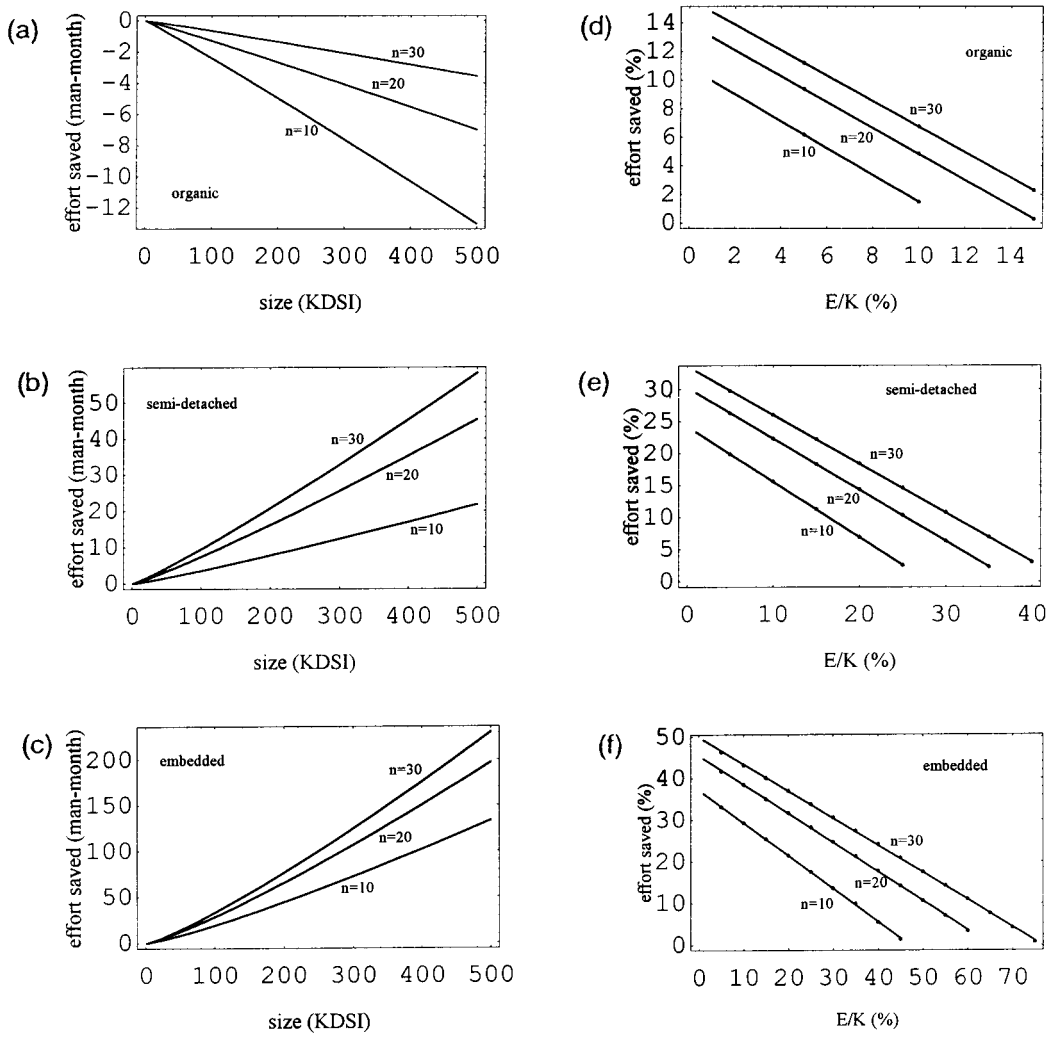


Figure 4. (a), (b), (c) The relationship between the maintenance effort savings and size; 1981 COCOMO with $ACT_i = 0{\cdot}1$, $E/K = 0{\cdot}2$, s_i *follows MP1 (uniform-like) distribution, and* e_i *is assumed to be uniformly distributed. (d), (e), (f) The relationship between maintenance effort saved and the* E/K *ratio; 1981 COCOMO with* $ACT_i = 0{\cdot}1$, e_i *follows MP1 distribution*

semi-detached mode systems, compared with a saving of 23 man-months in embedded mode systems. The ID can save even more if the system is 200 KDSI large—nine man-months and 45 man-months for semi-detached mode systems and embedded mode systems, respectively. However, as shown in Figure 4(a), ID may not be that beneficial in developing organic mode systems; in fact, when the E/K ratio, here 0·2, exceeds the CERs (refer to Table 5), an increase of maintenance effort is incurred.

The amount of effort saved also increases with the number of increments. For a system

Table 5. The CERs for the three different distributions of the extra code

	CER (%)								
	Organic			Semi-detached			Embedded		
Extra code (e_i)	$n = 10$	$n = 20$	$n = 30$	$n = 10$	$n = 20$	$n = 30$	$n = 10$	$n = 20$	$n = 30$
MP1	11·59	15·33	17·58	27·98	37·84	43·96	46·77	64·73	76·23
MP2	11·59	15·33	17·57	27·93	37·76	43·85	46·54	64·28	75·59
MP3	11·56	15·28	17·51	27·45	36·90	42·67	44·05	59·45	68·98

100 KDSI large with 20 increments and 0·2 of the E/K ratio, ID can save 7·5 man-months and 9·6 man-months of maintenance effort annually in semi-detached mode systems and embedded mode systems, respectively. The maintenance effort saved becomes even larger when the number of increments is 30 (28·6 man-months for semi-detached mode systems and 33·4 man-months for embedded mode systems).

We next address research questions 2: what is the ideal range for the number of increments of an ID system? The number of increments affects, either directly or indirectly, most of the important system development considerations, such as the criteria for system decomposition, project planning and monitoring, resource allocation and management, and the timing of the product delivery. To determine the ideal range for the number of increments of an ID system, we look for systems that allow a larger cost-effective extra code ratio among a set of similar systems with a varying number of increments. Figure 5(a) shows the functional relationship between the CERs and the number of increments. The data were derived from the 1981 COCOMO model by modelling the size distribution of the extra code (e_i) as MP1 distribution. The pattern of the scattered plots (CER, n) reveals two points: (1) the increase of CER is spectacular when the number of increments is in the range of five to 30, and (2) the speed of the increase of CER is largest in embedded mode systems. In other words, the increase of CER is very sensitive to the increase in number of increments in large and complex systems. These two points imply that there

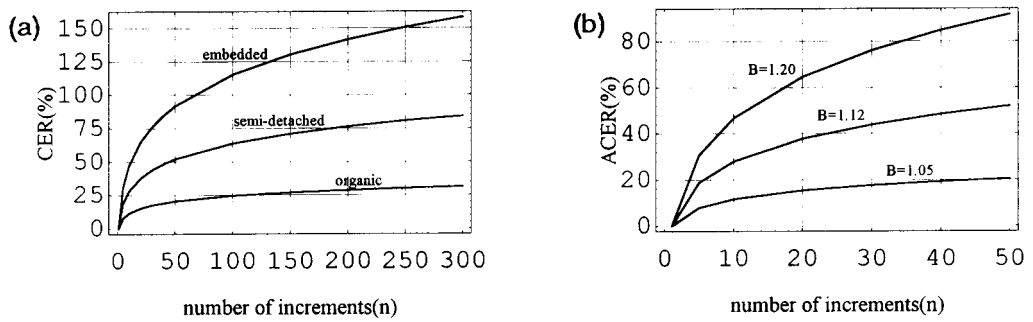


Figure 5. (a) The relationship between the CER and total number of increments. (b) The relationship between the ACER and total number of increments

are increased benefits for ID in large and complex systems and where the number of increments is between five and 30.

The increase of the CER indicates that ID is beneficial for large numbers of increments. However, the increase is asymptotically approaching a limit when the number of increments approaches infinity. Larger numbers of increments imply a preference towards smaller increments. As the figure shows, the organic mode curve begins to become flat when the number of increments is around 50. The semi-detached and embedded mode curves become flat when the numbers of increments are around 100 and 200, respectively. In reality, it is unusual to have a very large number of increments. Also, large numbers of increments create configuration management problems which may increase maintenance effort. The results give us a reference for the trade-off between the overall maintenance effort and the overhead resulting from managing large numbers of increments. By incorporating other factors, such as the characteristics of the problem itself or customers' specific system-application pattern, we suggest an ideal range for number of increments is from five to 20.

The data derived from COCOMO 2.0 reveal similar results. Figure 5(b) shows the functional relationship between adapted cost-effective extra code ratio (ACER) and the number of increments. ACER is defined as a ratio of the size of the total maintained extra code relative to the size of the total maintained non-extra code due to the fact that COCOMO 2.0 uses adapted source lines of code as the size measure. As illustrated in the figure, the distribution pattern of the scattered plots (ACER, n) is similar to that of (CER, n). Larger numbers of increments allow a larger percentage of the extra code of the total maintained source code.

Finally, the last research question is: how much extra code is allowed if ID is to be beneficial compared with MD in the context of software maintenance? To answer this question, we adjust the amount of the extra code (in 1981 COCOMO) and the maintained extra code (in COCOMO 2.0) up to a point so that the resulting maintenance effort of an ID system is equal to or less than that of the comparable MD system. Table 5 lists the cost-effective extra code ratios for some chosen ID systems. The data were derived from 1981 COCOMO by assuming that the non-extra code (k_i) is uniformly distributed. As shown in the table, the CERs range from 11·56% to 17·58% for organic mode systems, 27·45% to 43·96% for semi-detached mode systems and 44·05% to 76·23% for embedded mode systems. This means that if a 10-increment semi-detached ID system is to be more cost-effective than the comparable MD system, its extra code ratio should not be greater than 27·45%. To clarify, a 10-increment semi-detached MD system with 10 KDSI in size will need more maintenance effort than the comparable ID system, if the total size of the ID system is less than 12·745 KDSI. In this instance, any 10-increment semi-detached ID system with less than 2·745 KDSI of the extra code will be considered as cost-effective.

Also as illustrated in Table 5, CERs increase with the number of increments and the complexity of systems. For instance, the CERs for a semi-detached mode system with 10, 20 and 30 increments are approximately equal to 27·79%, 37·5% and 43·49%, respectively. The breaking down of a system into more increments will inevitably create more extra code. Thus, the trade-off between number of increments and amount of the resulting extra code is an important issue, and it will be one of the major concerns of our further research. Results show, for instance, that building a 10-increment embedded

mode system can allow approximately 45·79% of the extra code ratio. This is 18% (45·79%–27·79%) more than that of the semi-detached systems.

The introduction of extra code results in a loss of maintenance effort savings. The saving of maintenance effort is expected to decrease with the increase of extra code ratio (E/K). The results in Figures 4(d), 4(e) and 4(f) give us a clear picture of the trend of the loss of maintenance effort savings. As the figures show, the savings decrease quasi-linearly with the E/K ratio. However, systems with a large number of increments are less sensitive to the increase of the E/K ratio; for three modes of systems, the slopes of the 30-increment lines are less steep than those of the 20-increment and the 10-increment lines. This implies that, for an increase of the same amount of extra code, systems with a smaller number of increments will have less maintenance effort savings than those with a larger number of increments.

Figure 6 shows the relationship between $\hat{E}/\hat{K}$ and the scaling factor B. The data were derived by assuming that SAP_i and $\hat{k}_i$ are uniformly distributed. As shown in the figure, large scaling factors and/or number of increments allow a larger percentage of extra code in the total maintained code. In other words, ID systems with larger values of the $\hat{E}/\hat{K}$ ratio can tolerate more unstable extra code. However, one should strive for stabilizing extra code. Extra code that implements volatile requirements increases future maintenance efforts of ID systems and should be carefully handled up front in the formulation of increments.

6. CONCLUSIONS AND FUTURE WORK

We have utilized an analytical parametric study to assess the impact of ID on maintenance effort. ID was compared with MD from the standpoint of software maintenance, and two impact parameters, increment size and extra code, were modelled. Based on this analysis, the following conclusions are drawn:

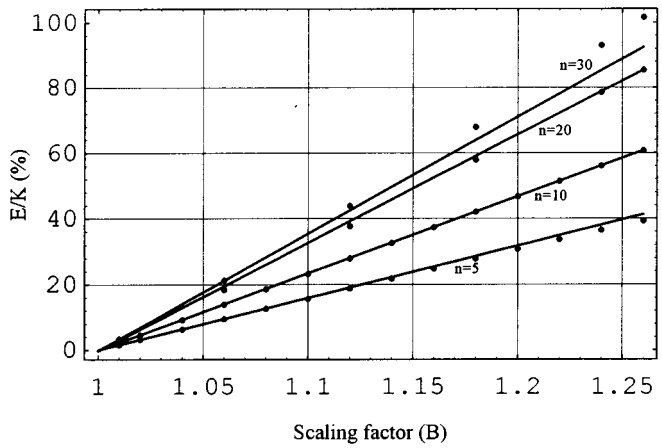


Figure 6. The relationship between $\hat{E}/\hat{K}$ *and the scaling factor* B*; COCOMO 2·0 with* $SAP_i = 0{\cdot}2$ *and* $\hat{k}_i$ *is uniformly distributed*

(1) ID is most beneficial for large, complex systems in terms of software maintenance. For instance, a 20-increment embedded mode ID system with 0·2 of the E/K ratio can save about one-third of the maintenance effort relative to a comparable MD system.
(2) The increase of the cost-effective extra code ratio (CER) and adapted cost-effective extra code ratio (ACER) with the number of increments indicates that ID is beneficial for large numbers of increments. However, this benefit is upper-bounded by a limit. On the other hand, a large number of increments may create configuration management problems that increase maintenance effort. We suggest that an ideal range for the number of increments is from five to 20.
(3) The largest allowed amount of the extra code for an ID system to be more cost-effective than a comparable MD system is represented as the cost-effective extra code ratio CER and adapted cost-effective extra code ratio ACER. We conclude from this study that the CERs range from 27·45% to 43·96% for semi-detached mode systems and from 44·05% to 76·23% for embedded mode systems. The values of ACER range from 11·60% ($B = 1{\cdot}05$ and number of increments = 10) to 76·23% ($B = 1{\cdot}20$ and number of increments = 30).

Based upon the results of this study, clustering requirements to form increments are critical from the perspective of software maintenance, since they directly influence the relative size of the increments and the amount of extra code. Thus, a good heuristic of 'design-for-maintenance' for requirements clustering and implementation of increments is needed. We plan to refine the requirements clustering algorithm (Hsia and Young, 1988) further, based on the results of this study.

Further research is needed to support these results through real-world data in specific application domains using either (1) historical data from commercial companies or (2) empirical data obtained by conducting well-controlled experiments. The distribution models for the size and the extra code of the increments need to be further validated with real-world data.

Another question that deserves further study is 'how much development effort is needed when employing ID in practice?' This question can be studied using a similar approach. We expect that ID will require higher up-front development effort due to the effort spent in partitioning a system into a set of loosely-coupled increments. However, the total development effort of an ID system may be less than that of a comparable MD system as long as the following two conditions are satisfied: (1) the system exhibits characteristics of diseconomies of scale; and (2) the amount of extra code is below a threshold‡. A similar parametric analysis can be performed to provide a more detailed comparison between ID and MD from the perspective of development effort.

‡ The value of the threshold, however, differs from the cost-effective extra code ratio derived from this study.

APPENDIX A

Notation

The notation that we use in this study is listed below. Note that we use capital letters to represent numbers of KDSI and lower-case letters to represent a ratio or a fraction.

ACT_{MD}	The ACT of a MD system.
S	The total size of an ID system.
S_i	The total size of increment i.
s_i	The fraction of the total source code of an ID system that is in increment i.
K	The total size of the non-extra code of all the increments in an ID system.
K_i	The size of the non-extra code of increment i.
k_i	The fraction of the total non-extra code of an ID system that is in increment i, i.e., $k_i = K_i/K$.
E	The total size of the extra code of all the increments in an ID system.
E_i	The size of the extra code of increment i.
e_i	The fraction of the extra code within increment i, namely $e_i = E_i/S_i$.
K	The total size of the adapted non-extra code of all the increments in an ID system.
$\hat{K}_i$	The size of the adapted non-extra code of increment i.
$\hat{k}_i$	The fraction of the total adapted non-extra code of an ID system that is in increment i, i.e., $\hat{k}_i = \hat{K}_i/\hat{K}$.
$\hat{E}$	The size of the total adapted extra code in an ID system.
$\hat{E}_i$	The size of the adapted extra code of increment i.
$\hat{e}_i$	The fraction of the adapted extra code within increment i, namely $\hat{e}_i = \hat{E}_i/\hat{S}_i$.

APPENDIX B

Derivation of equations

In an incrementally delivered software system, each increment is a semi-independent system. Maintenance of an increment will not affect the functionality of the other increments. Each increment is treated like a normal MD system in which the basic COCOMO maintenance model can be applied. The maintenance effort of an ID system is thus the arithmetic sum of the individual maintenance effort of all the increments within a system.

$$MM_{\mathrm{AM}} = \sum_{i=1}^{n} (MM_{\mathrm{AM}})_i = \sum_{i=1}^{n} ACT_i\,(MM_{\mathrm{D}})_i = \sum_{i=1}^{n} ACT_i\,(aS_i^b) \tag{2}$$

where n is the number of increments; ACT_i is the annual change traffic of increment i; $(MM_{\mathrm{D}})_i$ is the development effort in man-months of increment i; S_i is the size of increment i.

However, the model should be modified to represent correctly the maintenance effort of Case I as discussed in Section 3. In that case, only the change traffic of the first repair is counted among many increments which undergo similar software maintenance. As such, the 1981 basic COCOMO maintenance model should be reformulated as

$$(MM)_{\mathrm{AM}} = \sum_{i=1}^{n} (u_i \times ACT_i)\ (aS_i^b) \tag{3}$$

where $0 \leqslant u_i \leqslant 1$ is the fraction of the changed source instructions within increment i that is first repaired in a specific maintenance activity during a year.

Similarly, the COCOMO 2.0 maintenance model can be formulated as

$$MM = \sum_{i=1}^{n} A(ASLOC_i \times SAP_i)^B \tag{4}$$

The annual maintenance effort of an ID system, according to the 1981 COCOMO model, is determined by three parameters: u_i, ACT_i and S_i. The fraction of the extra code (e_i) and the non-extra code (k_i) also affects the maintenance effort. An ID system with a larger percentage of the extra code will incur more maintenance effort than a comparable MD system does.

For the purpose of comparison, we will simply assume that the annual change traffic of all of the increments is identical and equal to that of the comparable MD system, namely $ACT_i = ACT_{\mathrm{MD}}$ for $1 \leqslant i \leqslant n$. To simplify the comparison further, we will assume that $u_i = 1.0$ for $1 \leqslant i \leqslant n$; the worst case in which all the changes to each increment are assumed to be unique. This assumption, although biased against ID, provides us with a broader space of confidence about the results.

The COCOMO 2.0 maintenance effort estimate for a given value of *ASLOC* is determined by four parameters: SAP_i, $ASLOC_i$, $\hat{k}_i$ and $\hat{e}_i$. To compare the required maintenance effort of a delivered software system developed by two different approaches, MD and ID, we need to describe or model these parameters. Two parameters will be modelled in both the 1981 COCOMO and COCOMO 2.0. They are 'increment size' and 'the size of the extra code' in 1981 COCOMO and 'the size of the adapted source code' and 'the size of adapted extra code' in COCOMO 2.0.

Increment size

The size of an increment, say i, can be defined as a fraction of the overall size of all the increments within an ID system, namely $S_i = S \times s_i$, in which s_i can be modelled as $y(i)/\Sigma_{i=1}^{n} y(i)$ or $\tilde{y}(i)$. The 1981 COCOMO maintenance effort estimation equation in terms of the size distribution of the increments becomes

$$(MM)_{\mathrm{AM}} = \sum_{i=1}^{n} ACT_i \times a\ S_i^b = \sum_{i=1}^{n} ACT_i \times a(S \times \tilde{y}(i))^b \tag{5}$$

Similarly in COCOMO 2.0, the size of the adapted source code of increment i ($ASLOC_i$) can be defined as a fraction of the size of the total adapted source code of the whole ID system (ASLOC). That is, $ASLOC_i = ASLOC \times \hat{y}(i)$, where $\hat{y}(i)$ is modelled as $y(i)/\Sigma_{i=1}^{n} y(i)$.

The COCOMO 2.0 maintenance effort estimation equation is

$$MM = \sum_{i=1}^{n} A(ASLOC_i \times SAP_i)^B = \sum_{i=1}^{n} A \times SAP_i^B \times (ASLOC \times \hat{y}(i))^B \tag{6}$$

Extra code

The size of the extra code of an increment is defined as a fraction of the total size of the increment, namely

$$E_i = e_i \times S_i = e_i(K_i + E_i)$$

Substituting, we obtain

$$E_i\,(1 - e_i) = K_i \times e_i$$

$$E_i = \frac{K_i \times e_i}{1 - e_i} \tag{7}$$

The overall size of the extra code E within an ID system is the arithmetic sum of the size of the extra code of all the increments, namely

$$E = \sum_{i=1}^{n} E_i = \sum_{i=1}^{n} \frac{K_i \times e_i}{1 - e_i} = K \sum_{i=1}^{n} \frac{k_i \times e_i}{1 - e_i} \tag{8}$$

The ratio of the total size of the extra code over the total size of the non-extra code is

$$\frac{E}{K} = \sum_{i=1}^{n} \frac{k_i \times e_i}{1 - e_i}$$

The distribution of the extra code (e_i) can be modelled as $e_{\max} \times y(i)$, where $e_{\max}$ is the maximal e_i among all the increments. Thus,

$$\frac{E}{K} = \sum_{i=1}^{n} \frac{k_i \times e_{\max} \times y(i)}{1 - e_{\max} \times y(i)}$$

The 1981 COCOMO maintenance effort estimation equation in terms of the size distribution of the extra code becomes

$$(MM)_{\mathrm{AM}} = \sum_{i=1}^{n} ACT_i \times a\, S_i^b = \sum_{i=1}^{n} ACT_i \times a(K_i + E_i)^b$$

Substituting the E_i in equation (7), we get

$$(MM)_{\mathrm{AM}} = \sum_{i=1}^{n} ACT_i \times a\left(\frac{K_i}{1 - e_i}\right)^b = \sum_{i=1}^{n} ACT_i \times a\left(\frac{K \times k_i}{1 - e_{\max} \times y(i)}\right)^b \tag{9}$$

The value of $e_{\max}$ can be determined from the ratio E/K.

Following the same procedure of deriving equation (9), we get the COCOMO 2.0 maintenance effort estimation equation of an ID system:

$$MM = \sum_{i=1}^{n} A \times SAP_i^B \times \left(\frac{\hat{K}_i}{1 - \hat{e}_i}\right)^B = \sum_{i=1}^{n} A \times SAP_i^B \times \left(\frac{\hat{K} \times \hat{k}_i}{1 - \hat{e}_{\max} \times \hat{y}(i)}\right)^B \tag{10}$$

where $\hat{K}$ is the total non-extra source lines of code that is maintained; $\hat{e}_i = \dfrac{\hat{E}_i}{ASLOC_i}$; and $\hat{k}_i$ is the fraction of the total maintained non-extra code of an ID system that is in increment i.

References

Abbott, K. R. (1997) 'Product development: a chunk at a time', *Proceedings of the 8th IEEE International Workshop on Software Technology and Engineering Practice Incorporating Computer Aided Software Engineering*, IEEE Computer Society, Los Alamitos CA.

Alavi, M. (1984) 'An assessment of the prototyping approach to information systems development', *Communications of the ACM*, **27**(6), 556–563.

Banker, R. D. and Kemerer, C. F. (1989) 'Scale economies in new software development', *IEEE Transactions on Software Engineering*, **SE-15**(10), 1199–1205.

Basili, V. R. and Reiter, R. W., Jr. (1981) 'A controlled experiment quantitatively comparing software development approaches', *IEEE Transactions on Software Engineering*, **SE-7**(3), 299–320.

Boehm, B. W. (1981a) *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs NJ, 767pp.

Boehm, B. W. (1981b) 'An experiment in small-scale application software engineering', *IEEE Transactions on Software Engineering*, **SE-7**(5), 482–493.

Boehm, B. W., Gray, T. E. and Seewaldt, T. (1984) 'Prototyping versus specifying: a multiproject experiment', *IEEE Transactions on Software Engineering*, **SE-10**(3), 290–302.

Boehm, B., Clark, B., Horowitz, E., Westland, C., Madachy, R. and Selby, R. (1995) 'Cost models for future software life cycle processes: COCOMO 2.0', *Annals of Software Engineering*, **1**, 57–94.

Buck, R. D. and Dobbins, J. H. (1984) 'Application of software inspection methodology in design and code', *Proceedings of the Software Validation, Inspection-testing-verification-alternatives symposium*, North Holland, Amsterdam, pp. 41–56.

Conte, S. D., Dunsmore, H. E. and Shen, V. Y. (1986) *Software Engineering Metrics and Models*, Benjamin/Cummings, Menlo Park CA, 396pp.

Eckhard, D. E., Caglayan, A. K., Knight, J. C., Lee, L. D., McAllister, D. F., Vouk, M. A. and Kelly, J. P. J. (1991) 'An experimental evaluation of software redundancy as a strategy for improving reliability', *IEEE Transactions on Software Engineering*, **SE-17**(7), 692–702.

Gilb, T. (1988) *Principles of Software Engineering Management*, Addison-Wesley, Reading MA, 442pp.

Hough, D. (1993) 'Rapid delivery: an evolutionary approach for application development', *IBM Systems Journal*, **32**(3), 397–419.

Hsia, P., Yaung, A. T. and Jiam, S. H. (1986) 'Requirements clustering for incremental construction of software systems', in *Proceedings COMPSAC'86*, IEEE Computer Society Press, Washington DC, pp. 204–211.

Hsia, P. and Yaung, A. T. (1988) 'Another approach to system decomposition: requirements clustering', in *Proceedings COMPSAC'88*, IEEE Computer Society, New York NY, pp. 75–82.

Jiam, S. H. (1985) '*Operational approach vs conventional approach—a case study in software engineering*', Masters Project, Department of Computer Science and Engineering, The University of Texas at Arlington TX, 160pp.

Leung, H. K. N. and White, L. (1990) 'A study of integration testing and software regression at the integration level', in *Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society Press, Washington DC, pp. 290–301.

Linger, R. C. (1994) 'Cleanroom process model', *IEEE Software*, **11**(2), 50–58.

Mabson, G. E., Wharram, G. E., Tennyson, R. C. and Hansen, J. S. (1984) 'On the compressive strength of graphite composite laminates containing interlaminar flaws', *Polymer Plastics Technology in Engineering*, **22**, 99–113.

Mills, H. D., Dyer, M. and Linger, R. C. (1987) 'Cleanroom software engineering', *IEEE Software*, **4**(5), 19–25.

Miyano, Y. M., Kanemitsu, M., Kunio, T. and Kuhn, H. A. (1986) 'Role of matrix resin on fracture strengths of unidirectional CFRP', *Journal of Composite Materials*, **20**, 520–538.

Pittman, M. (1993) 'Lessons learned in managing object-oriented development', *IEEE Software*, **10**(1), 43–53.

Schneidewind, N. F. (1989) 'Software maintenance: the need for standardization', *Proceedings of the IEEE*, **77**(4), 618–624.

Tran, P. and Galka, R. (1991) 'On incremental delivery with functionality', in *Tenth Annual International Phoenix Conference on Computers and Communications*, IEEE Computer Society Press, Los Alamitos CA, pp. 369–375.

Authors' biographies:

Pei Hsia is a Professor of Computer Science and Engineering at the University of Texas at Arlington. He is also the Director of the Software Engineering Center for Telecommunications. His research interests include requirements engineering, concurrent software engineering, incremental delivery and software testing. His email address is: hsia@cse.uta.edu

Chih-Tung Hsu received his B.A. and M.S. degrees in engineering in Taiwan, and an M.S. in computer science from the University of Texas at Arlington, where he is currently a Ph.D. candidate. His research interests include software requirements specification, object-oriented software testing, incremental delivery and concurrent software engineering.

David C. Kung is an Associate Professor of Computer Science and Engineering at the University of Texas at Arlington. He received his M.S. and Ph.D. degrees in computer science from the Norwegian Institute of Technology, and in 1990, he worked as a staff software scientist at International Software Systems, Inc. His research interests are in real-time systems and object-oriented systems.

Eric Byrne is a Consultant with Global Consultants, Inc., where he is involved with software quality assurance and process improvement efforts. Formerly he was an Assistant Professor at the University of Texas at Arlington. His research interests include software maintenance, software engineering, software processes, software quality assurance (SQA), and program comprehension. Eric received his B.S. degree in physics and computer science from the University of Nebraska at Lincoln, and his M.S. and Ph.D. degrees from Kansas State University at Manhattan in Kansas.